APPENDIX A

```
network monitor/defender
//
// Has two operating modes: if MONITOR is defined, it monitors the network
// instead of defending against DDoS attacks.
//
// ICMP_RATE specifies how many ICMP packets allowed per second. Default is
// 500. UDP_NF_RATE specifies how many non-fragmented UDP (and other non-TCP
// non-ICMP) packets allowed per second. Default is 3000. UDP_F_RATE specifies
// how many fragmented UDP (and other non-TCP non-ICMP) packets allowed per
// second. Default is 1000. All the SNIFF rates specify how many bad packets
// sniffed per second.
//
// For example, if MONITOR is not defiend, and all SNIFF rates are 0, then the
// configuration defends against DDoS attacks, but does not report bad
// packets.
//
// can read:
//   - tcp_monitor: aggregate rates of different TCP packets
//   - ntcp_monitor: aggregate rates of different non TCP packets
//   - icmp_unreach_counter: rate of ICMP unreachable pkts
//   - tcp_ratemon: incoming and outgoing TCP rates, grouped by non-local hosts
//   - ntcp_ratemon: incoming UDP rates, grouped by non-local hosts
//
// Note: handles full fast ethernet, around 134,500 64 byte packets, from
// attacker.
//
//
// TODO:
//   - fragmented packet monitor

#ifndef ICMP_RATE
#define ICMP_RATE        500
#endif

#ifndef UDP_NF_RATE
#define UDP_NF_RATE      2000
#endif

#ifndef UDP_F_RATE
#define UDP_F_RATE       1000
#endif

#ifndef SUSP_SNIFF
#define SUSP_SNIFF       100     // # of suspicious pkts sniffed per sec
```

```
#endif

#ifndef TCP_SNIFF
#define TCP_SNIFF  100     // # of TCP flood pkts sniffed per sec
#endif

#ifndef ICMP_SNIFF
#define ICMP_SNIFF         75      // # of ICMP flood pkts sniffed per sec
#endif

#ifndef UDP_NF_SNIFF
#define UDP_NF_SNIFF       75      // # of non-frag UDP flood pkts sniffed per sec
#endif

#ifndef UDP_F_SNIFF
#define UDP_F_SNIFF        75      // # of frag UDP flood pkts sniffed per sec
#endif

#include "if.click"

#include "sampler.click"

#include "sniffer.click"
ds_sniffer :: Sniffer(mazu_ds);
syn_sniffer :: Sniffer(mazu_syn);
tcp_sniffer :: Sniffer(mazu_tcp);
ntcp_sniffer :: Sniffer(mazu_ntcp);

#include "synkill.click"
#ifdef MONITOR
tcpsynkill :: SYNKill(true);
#else
tcpsynkill :: SYNKill(false);
#endif


//
// discards suspicious packets
//

#include "ds.click"
ds :: DetectSuspicious(01);

from_world -> ds;
ds [0] -> is_tcp_to_victim :: IPClassifier(tcp, -);
```

```
#ifdef MONITOR
ds [1] -> ds_split :: RatedSampler(SUSP_SNIFF);
#else
ds [1] -> ds_split :: RatedSplitter(SUSP_SNIFF);
#endif

ds_split [1] -> ds_sniffer;
ds_split [0]
#ifdef MONITOR
  -> is_tcp_to_victim;
#else
  -> Discard;
#endif

//
// monitor TCP ratio
//

#include "monitor.click"
tcp_ratemon :: TCPTrafficMonitor;

is_tcp_to_victim [0] -> tcp_monitor :: TCPMonitor -> [0] tcp_ratemon;
from_victim -> is_tcp_to_world :: IPClassifier(tcp, -);
is_tcp_to_world [0] -> [1] tcp_ratemon;

//
// enforce correct TCP ratio
//

check_tcp_ratio :: RatioShaper(1,2,40,0.2);
tcp_ratemon [0] -> check_tcp_ratio;

#ifdef MONITOR
check_tcp_ratio [1] -> tcp_split :: RatedSampler(TCP_SNIFF);
#else
check_tcp_ratio [1] -> tcp_split :: RatedSplitter(TCP_SNIFF);
#endif

tcp_split [1] -> tcp_sniffer;
tcp_split [0]
#ifdef MONITOR
  -> [0] tcpsynkill;
#else
  -> Discard;
#endif
```

```
//
// prevent SYN bomb
//

check_tcp_ratio [0] -> [0] tcpsynkill;
tcp_ratemon [1] -> [1] tcpsynkill;

tcpsynkill [0] -> to_victim_s1;
tcpsynkill [1] -> to_world;

tcpsynkill [2]
#ifdef MONITOR
  -> syn_sniffer;
Idle -> to_victim_prio;
#else
  -> tcpsynkill_split :: Tee(2)
tcpsynkill_split [0] -> to_victim_prio;
tcpsynkill_split [1] -> syn_sniffer;
#endif

//
// monitor all non TCP traffic
//

ntcp_ratemon :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, false);
is_tcp_to_victim [1] -> ntcp_monitor :: NonTCPMonitor -> ntcp_t :: Tee(2);
ntcp_t [0] -> [0] ntcp_ratemon [0] -> Discard;
ntcp_t [1] -> [1] ntcp_ratemon;

//
// rate limit ICMP traffic
//

ntcp_ratemon [1] -> is_icmp :: IPClassifier(icmp, -);
is_icmp [0] -> icmp_split :: RatedSplitter (ICMP_RATE);

icmp_split [1] -> to_victim_s2;
icmp_split [0] -> icmp_sample :: RatedSampler (ICMP_SNIFF);

icmp_sample [1] -> ntcp_sniffer;
icmp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif
```

```
//
// rate limit other non TCP traffic (mostly UDP)
//

is_icmp [1] -> is_frag :: Classifier(6/0000, -);

is_frag [0] -> udp_split :: RatedSplitter (UDP_NF_RATE);

udp_split [0] -> udp_sample :: RatedSampler (UDP_NF_SNIFF);
udp_sample [1] -> ntcp_sniffer;
udp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

is_frag [1] -> udp_f_split :: RatedSplitter (UDP_F_RATE);

udp_f_split [0] -> udp_f_sample :: RatedSampler (UDP_F_SNIFF);
udp_f_sample [1] -> ntcp_sniffer;
udp_f_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

//
// further shape non-TCP traffic with ICMP dest unreachable packets
//

is_tcp_to_world [1] -> is_icmp_unreach :: IPClassifier(icmp type 3, -);
is_icmp_unreach [1] -> to_world;
is_icmp_unreach [0]
  -> icmp_unreach_counter :: Counter;

#ifndef MONITOR

icmp_unreach_counter -> icmperr_sample :: RatedSampler (UNREACH_SNIFF);
icmperr_sample [1] -> ntcp_sniffer;
icmperr_catcher :: AdaptiveShaper(.1, 50);
udp_split [1] -> [0] icmperr_catcher [0] -> to_victim_s2;
udp_f_split [1] -> [0] icmperr_catcher;
icmperr_sample [0] -> [1] icmperr_catcher [1] -> to_world;
```

```
#else

udp_split [1] -> to_victim_s2;
udp_f_split [1] -> to_victim_s2;
icmp_unreach_counter [0] -> to_world;

#endif
```

== if.click

```
//
// input/output ethernet interface for router
//
// this configuration file leaves the following elements to be hooked up:
//
// from_victim:    packets coming from victim
// from_world:     packets coming from world
// to_world:       packets going to world
// to_victim_prio: high priority packets going to victim
// to_victim_s1:   best effort packets going to victim, tickets = 4
// to_victim_s2:   best effort packets going to victim, tickets = 1
//
// see bridge.click for a simple example of how to use this configuration.

// victim network is 1.0.0.0/8 (eth1, 00:C0:95:E2:A8:A0)
//  world network is 2.0.0.0/8 (eth2, 00:C0:95:E2:A8:A1) and
//                 3.0.0.0/8 (eth3, 00:C0:95:E1:B5:38)

// ethernet input/output, forwarding, and arp machinery

tol :: ToLinux;
t :: Tee(6);
t[5] -> tol;

arpq1_prio :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s1 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s2 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
ar1 :: ARPResponder(1.0.0.1/32 00:C0:95:E2:A8:A0);
arpq2 :: ARPQuerier(2.0.0.1, 00:C0:95:E2:A8:A1);
ar2 :: ARPResponder(2.0.0.1/32 00:C0:95:E2:A8:A1);
arpq3 :: ARPQuerier(3.0.0.1, 00:C0:95:E1:B5:38);
ar3 :: ARPResponder(3.0.0.1/32 00:C0:95:E1:B5:38);
```

```
psched :: PrioSched;
ssched :: StrideSched (4,1);

out1_s1 :: Queue(256) -> [0] ssched;
out1_s2 :: Queue(256) -> [1] ssched;
out1_prio :: Queue(256) -> [0] psched;
ssched -> [1] psched;
psched[0] -> to_victim_counter :: Counter -> todev1 :: ToDevice(eth1);

out2 :: Queue(1024) -> todev2 :: ToDevice(eth2);
out3 :: Queue(1024) -> todev3 :: ToDevice(eth3);

to_victim_prio :: Counter -> tvpc :: Classifier(16/01, -);
tvpc [0] -> [0]arpq1_prio -> out1_prio;
tvpc [1] -> Discard;

to_victim_s1 :: Counter -> tvs1c :: Classifier(16/01, -);
tvs1c [0] -> [0]arpq1_s1 -> out1_s1;
tvs1c [1] -> Discard;

to_victim_s2 :: Counter -> tvs2c :: Classifier(16/01, -);
tvs2c [0] -> [0]arpq1_s2 -> out1_s2;
tvs2c [1] -> Discard;

to_world :: Counter -> twc :: Classifier(16/02, 16/03, -);
twc [0] -> [0]arpq2 -> out2;
twc [1] -> [0]arpq3 -> out3;
twc [2] -> Discard;

from_victim :: GetIPAddress(16);
from_world :: GetIPAddress(16);

indev1 :: PollDevice(eth1);
c1 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                12/0800,
                -);
indev1 -> from_victim_counter :: Counter -> c1;
c1 [0] -> ar1 -> out1_s1;
c1 [1] -> t;
c1 [2] -> Strip(14) -> MarkIPHeader -> from_victim;
c1 [3] -> Discard;
t[0] -> [1] arpq1_prio;
t[1] -> [1] arpq1_s1;
t[2] -> [1] arpq1_s2;
```

```
indev2 :: PollDevice(eth2);
c2 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                12/0800,
                -);
indev2 -> from_attackers_counter :: Counter -> c2;
c2 [0] -> ar2 -> out2;
c2 [1] -> t;
c2 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c2 [3] -> Discard;
t[3] -> [1] arpq2;

indev3 :: PollDevice(eth3);
c3 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                12/0800,
                -);
indev3 -> c3;
c3 [0] -> ar3 -> out3;
c3 [1] -> t;
c3 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c3 [3] -> Discard;
t[4] -> [1] arpq3;

ScheduleInfo(todev1 10, indev1 1,
        todev2 10, indev2 1,
          todev3 10, indev3 1);
```

== sampler.click

```
elementclass RatedSampler {
 $rate |
  input -> s :: RatedSplitter($rate);
  s [0] -> [0] output;
  s [1] -> t :: Tee;
  t [0] -> [0] output;
  t [1] -> [1] output;
};

elementclass ProbSampler {
 $prob |
  input -> s :: ProbSplitter($prob);
  s [0] -> [0] output;
```

```
 s [1] -> t :: Tee;
 t [0] -> [0] output;
 t [1] -> [1] output;
};
```

== sniffer.click

```
// setup a sniffer device, with a testing IP network address
//
// argument: name of the device to setup and send packet to

elementclass Sniffer {
 $dev |
  FromLinux($dev, 192.0.2.0/24) -> Discard;

  input -> sniffer_ctr :: Counter
       -> ToLinuxSniffers($dev);
};

// note: ToLinuxSniffers take 2 us
```

== synkill.click

```
//
// SYNKill
//
// argument: true if monitor only, false if defend
//
// expects: input 0 - TCP packets with IP header to victim network
//          input 1 - TCP packets with IP header to rest of internet
//
// action:  protects against SYN flood by prematurely finishing the three way
//          handshake protocol.
//
// outputs: output 0 - TCP packets to victim network
//          output 1 - TCP packets to rest of internet
//          output 2 - control packets (created by TCPSYNProxy) to victim
//

elementclass SYNKill {
 $monitor |
  // TCPSYNProxy(MAX_CONNS, THRESH, MIN_TIMEOUT, MAX_TIMEOUT,
PASSIVE);
  tcpsynproxy :: TCPSYNProxy(128, 4, 8, 80, $monitor);
```

```
  input [0] -> [0] tcpsynproxy [0] -> [0] output;
  input [1] -> [1] tcpsynproxy [1] -> [1] output;
  tcpsynproxy [2]
    -> GetIPAddress(16)
    -> [2] output;
};
```

== ds.click

```
//
// DetectSuspicious
//
// argument: takes in the victim network address and mask. for example:
//    DetectSuspicious(121A0400%FFFFFF00)
//
// expects: IP packets.
//
// action: detects packets with bad source addresses;
//        detects direct broadcast packets;
//        detects ICMP redirects.
//
// outputs: output 0 push out accepted packets, unmodified;
//         output 1 push out rejected packets, unmodified.
//

elementclass DetectSuspicious {
  $vnet |

  // see http://www.ietf.org/internet-drafts/draft-manning-dsua-03.txt for a
  // list of bad source addresses to block out. we also block out packets with
  // broadcast dst addresses.

  bad_addr_filter :: Classifier(
    12/$vnet,                 // port  0: victim network address
    12/00,                    // port  1: 0.0.0.0/8 (special purpose)
    12/7F,                    // port  2: 127.0.0.0/8 (loopback)
    12/0A,                    // port  3: 10.0.0.0/8 (private network)
    12/AC10%FFF0,             // port  4: 172.16.0.0/12 (private network)
    12/C0A8,                  // port  5: 192.168.0.0/16 (private network)
    12/A9FE,                  // port  6: 169.254.0.0/16 (autoconf addr)
    12/C0000200%FFFFFF00,     // port  7: 192.0.2.0/24 (testing addr)
    12/E0%F0,                 // port  8: 224.0.0.0/4 (class D - multicast)
    12/F0%F0,                 // port  9: 240.0.0.0/4 (class E - reserved)
    12/00FFFFFF%00FFFFFF,     // port 10: broadcast saddr X.255.255.255
```

```
  12/0000FFFF%0000FFFF,      // port 11: broadcast saddr X.Y.255.255
  12/000000FF%000000FF,      // port 12: broadcast saddr X.Y.Z.255
  16/00FFFFFF%00FFFFFF,      // port 13: broadcast daddr X.255.255.255
  16/0000FFFF%0000FFFF,      // port 14: broadcast daddr X.Y.255.255
  16/000000FF%000000FF,      // port 15: broadcast daddr X.Y.Z.255
  9/01,                      // port 16: ICMP packets
  -);

 input -> bad_addr_filter;
 bad_addr_filter [0]  -> [1] output;
 bad_addr_filter [1]  -> [1] output;
 bad_addr_filter [2]  -> [1] output;
 bad_addr_filter [3]  -> [1] output;
 bad_addr_filter [4]  -> [1] output;
 bad_addr_filter [5]  -> [1] output;
 bad_addr_filter [6]  -> [1] output;
 bad_addr_filter [7]  -> [1] output;
 bad_addr_filter [8]  -> [1] output;
 bad_addr_filter [9]  -> [1] output;
 bad_addr_filter [10] -> [1] output;
 bad_addr_filter [11] -> [1] output;
 bad_addr_filter [12] -> [1] output;
 bad_addr_filter [13] -> [1] output;
 bad_addr_filter [14] -> [1] output;
 bad_addr_filter [15] -> [1] output;

 // ICMP rules: drop all fragmented and redirect ICMP packets

 bad_addr_filter [16]
   -> is_icmp_frag_packets :: Classifier(6/0000, -);
 is_icmp_frag_packets [1] -> [1] output;

 is_icmp_frag_packets [0]
   -> is_icmp_redirect :: IPClassifier(icmp type 5, -);
 is_icmp_redirect [0] -> [1] output;

 // finally, allow dynamic filtering of bad src addresses we discovered
 // elsewhere in our script.

 dyn_saddr_filter :: AddrFilter(SRC, 32);
 is_icmp_redirect [1] -> dyn_saddr_filter;
 bad_addr_filter [17] -> dyn_saddr_filter;
 dyn_saddr_filter [0] -> [0] output;
 dyn_saddr_filter [1] -> [1] output;

};
```

== monitor.click

```
//
// TCPTrafficMonitor
//
// expects: input 0 takes TCP packets w IP header for the victim network;
//          input 1 takes TCP packets w IP Header from the victim network.
// action:  monitors packets passing by
// outputs: output 0 - packets for victim network, unmodified;
//          output 1 - packets from victim network, unmodified.
//

elementclass TCPTrafficMonitor {
  // fwd annotation = rate of src_addr, rev annotation = rate of dst_addr
  tcp_rm :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, true);

  // monitor all TCP traffic to victim, monitor non-RST packets from victim
  input [0] -> [0] tcp_rm [0] -> [0] output;
  input [1] -> i1_tcp_rst :: IPClassifier(rst, -);
  i1_tcp_rst[0] -> [1] output;
  i1_tcp_rst[1] -> [1] tcp_rm [1] -> [1] output;
};
```

APPENDIX B

Appendix listing of additional Click modules ("elements").

```
ADAPTIVESHAPER(n)                                    ADAPTIVESHAPER(n)


NAME
       AdaptiveShaper - Click element

SYNOPSIS
       AdaptiveShaper(DROP_P, REPRESS_WEIGHT)

PROCESSING TYPE
       Push

DESCRIPTION
       AdaptiveShaper is a push element that shapes input traffic
       from input port 0 to output port  0.  Packets  are  shaped
       based  on "repressive" traffic from input port 1 to output
       port 1. Each repressive packet increases a  multiplicative
       factor  f  by  REPRESS_WEIGHT. Each input packet is killed
       instead of pushed out with f * DROP_P  probability.  After
       each dropped packet, f is decremented by 1.


EXAMPLES
ELEMENT HANDLERS
       drop_prob (read/write)
            value of DROP_P



       repress_weight (read/write)
            value of REPRESS_WEIGHT




SEE ALSO
       PacketShaper(n), RatioShaper(n)
```

APPENDIX B

ADAPTIVESPLITTER(n) ADAPTIVESPLITTER(n)

NAME

AdaptiveSplitter - Click element

SYNOPSIS

AdaptiveSplitter(RATE)

PROCESSING TYPE

Push

DESCRIPTION

AdaptiveSplitter attempts to split RATE number of packets per second for each address. It takes the fwd_rate annotation set by IPRateMonitor(n), and calculates a split probability based on that rate. The split probability attempts to guarantee RATE number of packets per second. That is, the lower the fwd_rate, the higher the split probability.

Splitted packets are on output port 1. Other packets are on output port 0.

EXAMPLES

```
AdaptiveSplitter(10);
```

SEE ALSO

IPRateMonitor(n)

APPENDIX B

ADDRFILTER(n) ADDRFILTER(n)

NAME

AddrFilter - Click element

SYNOPSIS

AddrFilter(DST/SRC, N)

PROCESSING TYPE

Push

DESCRIPTION

Filters out IP addresses given in write handler. DST/SRC specifies which IP address (dst or src) to filter. N is the maximum number of IP addresses to filter at any time. Packets passed the filter goes to output 0. Packets rejected by the filter goes to output 1.

AddrFilter looks at addresses in the IP header of the packet, not the annotation. It requires an IP header annotation (MarkIPHeader(n)).

EXAMPLES

AddrFilter(DST, 8)

Filters by dst IP address, up to 8 addresses.

ELEMENT HANDLERS

table ((read))
Dumps the list of addresses to filter and

add ((write))
Expects a string "addr mask duration", where addr is an IP address, mask is a netmask, and duration is the number of seconds to filter packets from this IP address. If 0 is given as a duration, filtering is removed. For example, "18.26.4.0 255.255.255.0 10" would filter out all packets with dst or source address 18.26.4.* for 10 seconds. New addresses push out old addresses if more than N number of filters already exist.

reset ((write))
Resets on write.

SEE ALSO

Classifier(n), MarkIPHeader(n)

APPENDIX B

ATTACKLOG(n) ATTACKLOG(n)

NAME

AttackLog - Click element; maintains a log of attack packets in SAVE_FILE.

SYNOPSIS

AttackLog(SAVE_FILE, INDEX_FILE, MULTIPLIER, PERIOD)

PROCESSING TYPE

Agnostic

DESCRIPTION

Maintains a log of attack packets in SAVE_FILE. Expects packets with ethernet headers, but with the first byte of the ethernet header replaced by an attack bitmap, set in kernel. AttackLog classifies each packet by the type of attack, and maintains an attack rate for each type of attack. The attack rate is the arrival rate of attack packets multiplied by MULTIPLIER.

AttackLog writes a block of data into SAVE_FILE once every PERIOD number of seconds. Each block is composed of entries of the following format:

delimiter (0s)	4 bytes	
time	4 bytes	
attack type		2 bytes
attack rate		4 bytes
ip header and payload (padded)	86 bytes	
	100 bytes	

Entries with the same attack type are written out together. A delimiter of 0xFFFFFFFF is written to the end of each block.

A circular timed index file is kept in INDEX_FILE along side the attacklog. See CircularIndex(n).

SEE ALSO

CircularIndex(n)

APPENDIX B

CIRCULARINDEX(n) CIRCULARINDEX(n)

NAME

CircularIndex - Click element; writes a timed circular index into a file.

SYNOPSIS

CircularIndex

DESCRIPTION

CircularIndex writes an entry into a circular index file periodically. The entry contains a 32 bit time stamp and a 64 bit offset into another file. The following functions are exported by CircularIndex.

int initialize(String FILE, unsigned PERIOD, unsigned WRAP) - Use FILE as the name of the circular file. Writes entry into circular file once every PERIOD number of seconds. WRAP is the number of writes before wrap around. If WRAP is 0, the file is never wrapped around.

void write_entry(long long offset) - Write entry into index file. Use offset as the offset in the entry.

SEE ALSO

GatherRates(n), MonitorSRC16(n)

APPENDIX B

DISCARDTODEVICE(n) DISCARDTODEVICE(n)

NAME

DiscardToDevice - Click element; drops all packets. gives skbs to device.

SYNOPSIS

DiscardToDevice(DEVICE)

PROCESSING TYPE

Agnostic

DESCRIPTION

Discards all packets received on its single input. Gives all skbuffs to specified device.

APPENDIX B

```
FILTERTCP(n)                                                FILTERTCP(n)


NAME
       FilterTCP - Click element

SYNOPSIS
       FilterTCP()

PROCESSING TYPE
       Push

DESCRIPTION
       Expects TCP/IP packets as input.
```

APPENDIX B

FROMTUNNEL(n) FROMTUNNEL(n)

NAME

FromTunnel - Click element

SYNOPSIS

FromTunnel(TUNNEL, SIZE, BURST)

PROCESSING TYPE

Push

DESCRIPTION

Grab packets from kernel KUTunnel element. TUNNEL is a /proc file in the handler directory of the KUTunnel element. SIZE specifies size of the buffer to use (if packet in kernel has larger size, it is dropped). BURST specifies the maximum number of packets to push each time FromTunnel runs.

EXAMPLES

FromTunnel(/proc/click/tunnel/config)

APPENDIX B

GATHERRATES(n) GATHERRATES(n)

NAME

GatherRates - Click element

SYNOPSIS

GatherRates(SAVE_FILE, INDEX_FILE, TCPMONITOR_IN, TCPMONITOR_OUT, MONITOR_PERIOD, SAVE_PERIOD);

PROCESSING TYPE

Agnostic

DESCRIPTION

Gathers aggregate traffic rates from TCPMonitor(n) element at TCPMONITOR_IN and TCPMONITOR_OUT.

Aggregate rates are gathered once every MONITOR_PERIOD number of seconds. They are averaged and saved to SAVE_FILE once every SAVE_PERIOD number of seconds. The following entry is written to SAVE_FILE for both incoming and outgoing traffic:

delimiter (0s)	4 bytes
time	4 bytes
type (0 for incoming traffic, 1 for outgoing traffic)	4 bytes
packet rate of tcp traffic	4 bytes
byte rate of tcp traffic	4 bytes
rate of fragmented tcp packets	4 bytes
rate of tcp syn packets	4 bytes
rate of tcp fin packets	4 bytes
rate of tcp ack packets	4 bytes
rate of tcp rst packets	4 bytes
rate of tcp psh packets	4 bytes
rate of tcp urg packets	4 bytes
packet rate of non-tcp traffic	4 bytes
byte rate of non-tcp traffic	4 bytes
rate of fragmented non-tcp traffic	4 bytes
rate of udp packets	4 bytes
rate of icmp packets	4 bytes
rate of all other packets	4 bytes
	72 bytes

After the two entries, an additional delimiter of 0xFFFFFFFF is written. SAVE_PERIOD must be a multiple of MONITOR_PERIOD.

A circular timed index is kept along side the stats file. See CircularIndex(n).

SEE ALSO

TCPMonitor(n) CircularIndex(n)

APPENDIX B

```
ICMPPINGENCAP(n)                                    ICMPPINGENCAP(n)


NAME
       ICMPPINGEncap - Click element

SYNOPSIS
       ICMPPINGEncap(SADDR, DADDR [, CHECKSUM?])

DESCRIPTION
       Encapsulates each incoming packet in a ICMP ECHO/IP packet
       with source address SADDR and destination  address  DADDR.
       The  ICMP  and IP checksums are calculated if CHECKSUM? is
       true; it is true by default.


EXAMPLES
         ICMPPINGEncap(1.0.0.1, 2.0.0.2)
```

APPENDIX B

KUTUNNEL(n) KUTUNNEL(n)

NAME

KUTunnel - Click element; stores packets in a FIFO queue that userlevel Click elements pull from.

SYNOPSIS

KUTunnel([CAPACITY])

PROCESSING TYPE

Push

DESCRIPTION

Stores incoming packets in a first-in-first-out queue. Drops incoming packets if the queue already holds CAPACITY packets. The default for CAPACITY is 1000. Allows user-level elements to pull from queue via ioctl.

ELEMENT HANDLERS

length (read-only)

Returns the current number of packets in the queue.

highwater_length (read-only)

Returns the maximum number of packets that have ever been in the queue at once.

capacity (read/write)

Returns or sets the queue's capacity.

drops (read-only)

Returns the number of packets dropped so far.

SEE ALSO

Queue(n)

APPENDIX B

NAME

Logger - Click element

SYNOPSIS

Logger(LOGFILE, INDEXFILE [, LOCKFILE, COMPRESS?, LOGSIZE, PACKETSIZE, WRITEPERIOD, IDXCOALESC, PACKETFREQ, MAXBUF-SIZE])

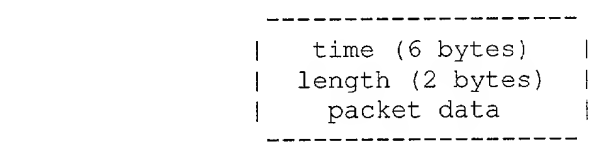
PROCESSING TYPE

Agnostic

DESCRIPTION

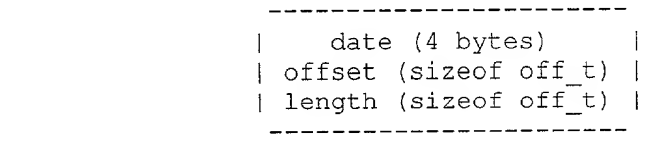
Has one input and one output.

Write packets to log file LOGFILE. A log file is a circular buffer containing packet records of the following form:



Time is the number of seconds and milliseconds since the Epoch at which a given packet was seen. Length is the length (in bytes) of the subsequent logged packet data. One or more packet records constitute one packet sequence.

INDEXFILE maintains control data for LOGFILE. It contains a sequence of sequence control blocks of the following form:



Date is a number of seconds since the Epoch. Offset points to the beginning of the packet sequence, i.e. to the earliest packet record having a time no earlier than date. Length is the number of bytes in the packet sequence. IDXCOALESC is the number of coalescing packets that a control block always cover. Default is 1024.

Sequence control blocks are always stored in increasing chronological order; offsets need not be in increasing order, since LOGFILE is a circular buffer.

COMPRESS? (true, false) determines whether packet data is logged in compressed form. Default is true.

APPENDIX B

LOGSIZE specifies the maximum allowable log file size, in KB. Default is 2GB. LOGSIZE=0 means "grow as necessary".

PACKETSIZE is the amount of packet data stored in the log. By default, the first 120 (128-6-2) bytes are logged and the remainder is discarded. Note that PACKETSIZE is the amount of data logged before compression.

Packet records are buffered in memory and periodically written to LOGFILE as a packet sequence. WRITEPERIOD is the number of seconds that should elapse between writes to LOGFILE. Default is 60. INDEXFILE is updated every time a sequence of buffered packet records is written to LOGFILE. The date in the sequence control block is the time of the first packet record of the sequence, with milliseconds omitted.

PACKETFREQ is an estimate of the number of packets per second that will be passing through Logger. Combined with WRITEPERIOD, this is a hint of buffer memory requirements. By default, PACKETFREQ is 1000. Since by default WRITEPERIOD is 60 and each packet record is at most 128 bytes, Logger normally allocates 7500KB of memory for the buffer. Logger will grow the memory buffer as needed up to a maximum of MAXBUFSIZE KB, at which point the buffered packet records are written to disk even if WRITEPERIOD seconds have not elapsed since the last write. Default MAXBUFSIZE is 65536 (64MB).

APPENDIX B

MONITORSRC16(n) MONITORSRC16(n)

NAME

MonitorSRC16 - Click element

SYNOPSIS

MonitorSRC16(SAVE_FILE, INDEX_FILE, MULTIPLIER, PERIOD, WRAP)

PROCESSING TYPE

Agnostic

DESCRIPTION

Examines src address of packets passing by. Collects statistics for each 16 bit IP address prefix. The following data structure is written to SAVE_FILE for every 16 bit IP address prefix every PERIOD number of seconds.

delimiter (0s)	(4 bytes)
time	(4 bytes)
addr	(4 bytes)
tcp rate	(4 bytes)
non tcp rate	(4 bytes)
percent of tcp	(1 byte)
percent of tcp frag	(1 byte)
percent of tcp syn	(1 byte)
percent of tcp fin	(1 byte)
percent of tcp ack	(1 byte)
percent of tcp rst	(1 byte)
percent of tcp psh	(1 byte)
percent of tcp urg	(1 byte)
percent of non tcp frag	(1 byte)
percent of udp	(1 byte)
percent of icmp	(1 byte)
reserved	(1 byte)
	32 bytes

TCP and non TCP rates are multiplied by MULTIPLIER. An additional delimiter of 0xFFFFFFFF is written at the end of a block of entries.

WARP specifies the number of writes before wrap-around. For example, if PERIOD is 60, WARP is 5, then every 5 minutes, the stats file wrap around.

A timed circular index is maintained along side the statistics file in INDEX_FILE. See CircularIndex(n).

SEE ALSO

CircularIndex(n)

APPENDIX B

RANDOMTCPIPENCAP(n) RANDOMTCPIPENCAP(n)

NAME

RandomTCPIPEncap - Click element

SYNOPSIS

RandomTCPIPEncap(DA BITS [DP SEQN ACKN CHECKSUM SA MASK])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a TCP/IP packet with random source address and source port, destination address DA, and control bits BITS. If BITS is -1, control bits are also generated randomly. If destination port DP, sequence number SEQN, or ack number ACKN is specified and non-zero, it is used. Otherwise, it is generated randomly for each packet. IP and TCP checksums are calculated if CHECKSUM is true; it is true by default. SEQN and ACKN should be in host order. SA and MASK are optional IP address; if they are specified, the source address is computed as ((random() & MASK) | SA).

EXAMPLES

RandomTCPIPEncap(1.0.0.2 4)

SEE ALSO

RoundRobinTCPIPEncap(n), RandomUDPIPEncap(n)

APPENDIX B

RANDOMUDPIPENCAP(n) RANDOMUDPIPENCAP(n)

NAME

RandomUDPIPEncap - Click element

SYNOPSIS

RandomUDPIPEncap(SADDR SPORT DADDR DPORT PROB [CHECKSUM?] [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a UDP/IP packet with source address SADDR, source port SPORT, destination address DADDR, and destination port DPORT. The UDP checksum is calculated if CHECKSUM? is true; it is true by default.

PROB gives the relative chance of this argument be used over others.

The RandomUDPIPEncap element adds both a UDP header and an IP header.

You can a maximum of 16 arguments. Each argument specifies a single UDP/IP header. The element will randomly pick one argument. The relative probabilities are determined by PROB.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
RandomUDPIPEncap(1.0.0.1 1234 2.0.0.2 1234 1 1,
                 1.0.0.2 1093 2.0.0.2 1234 2 1)
```

Will send about twice as much UDP/IP packets with 1.0.0.2 as its source address than packets with 1.0.0.1 as its source address.

SEE ALSO

Strip(n), UDPIPEncap(n), RoundRobinUDPIPEncap(n)

APPENDIX B

RATEWARN(n) RATEWARN(n)

NAME

RateWarn - Click element; classifies traffic and sends out warnings when rate of traffic exceeds specified rate.

SYNOPSIS

RateWarn(RATE, WARNFREQ)

PROCESSING TYPE

Push

DESCRIPTION

RateWarn has three output ports. It monitors the rate of packet arrival on input port 0. Packets are forwarded to output port 0 if rate is below RATE. If rate exceeds RATE, it sends out a warning packet WARNFREQ number of seconds apart on output port 2 in addition to forwarding all traffic through output port 1.

SEE ALSO

PacketMeter(n)

APPENDIX B

RATIOSHAPER(n) RATIOSHAPER(n)

NAME

RatioShaper - Click element

SYNOPSIS

RatioShaper(FWD_WEIGHT, REV_WEIGHT, THRESH, P)

PROCESSING TYPE

Push

DESCRIPTION

RatioShaper shapes packets based on fwd_rate_anno and rev_rate_anno rate annotations set by IPRateMonitor(n). If either annotation is greater than THRESH, and FWD_WEIGHT*fwd_rate_anno > REV_WEIGHT*rev_rate_anno, the packet is moved onto output port 1 with a probability of

```
min(1, P*(fwd_rate_anno*FWD_WEIGHT)/(rev_rate_anno*REV_WEIGHT))
```

FWD_WEIGHT, REV_WEIGHT, and THRESH are integers. P is a decimal between 0 and 1. Otherwise, packet is forwarded on output port 0.

EXAMPLES

```
RatioShaper(1, 2, 100, .2);
```

if fwd_rate_anno more than twice as big as rev_rate_anno, and both rates are above 100, drop packets with an initial probability of 20 percent.

ELEMENT HANDLERS

fwd_weight (read/write)
value of FWD_WEIGHT

rev_weight (read/write)
value of REV_WEIGHT

thresh (read/write)
value of THRESH

drop_prob (read/write)
value of P

SEE ALSO

Block(n), IPRateMonitor(n)

APPENDIX B

REPORTACTIVITY(n) REPORTACTIVITY(n)

NAME

ReportActivity - Click element

SYNOPSIS

ReportActivity(SAVE_FILE, IDLE)

PROCESSING TYPE

Agnostic

DESCRIPTION

Write into SAVE_FILE a 32 bit time value followed by an ASCII representation of that time stamp whenever a packet comes by. If IDLE number of seconds pass by w/o a packet, removes the file.

APPENDIX B

ROUNDROBINSETIPADDRESS(n) ROUNDROBINSETIPADDRESS(n)

NAME

RoundRobinSetIPAddress - Click element

SYNOPSIS

RoundRobinSetIPAddress(ADDR [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Set the destination IP address annotation of each packet with an address chosen from the configuration string in round robin fashion. Does not compute checksum (use SetIPChecksum(n) or SetUDPTCPChecksum(n)) or encapsulate the packet with headers (use RoundRobinUDPIPEncap(n) or RoundRobinTCPIPEncap(n) with bogus address).

EXAMPLES

```
RoundRobinUDPIPEncap(2.0.0.2 0.0.0.0 0 0 0)
  -> RoundRobinSetIPAddress(1.0.0.2, 1.0.0.3, 1.0.0.4)
  -> StoreIPAddress(12)
  -> SetIPChecksum
  -> SetUDPTCPChecksum
```

this configuration segment places an UDP header onto each packet, with randomly generated source and destination ports. The destination IP address is 2.0.0.2, the source IP address is 1.0.0.2, or 1.0.0.3, or 1.0.0.4. Both IP and UDP checksum are computed.

SEE ALSO

RoundRobinUDPIPEncap(n), RoundRobinTCPIPEncap(n), UDPIPEncap(n) , SetIPChecksum(n), SetUDPTCPChecksum(n), SetIPAddress(n), StoreIPAddress(n)

APPENDIX B

ROUNDROBINTCPIPENCAP(n) ROUNDROBINTCPIPENCAP(n)

NAME

RoundRobinTCPIPEncap - Click element

SYNOPSIS

RoundRobinTCPIPEncap(SA DA BITS [SP DP SEQN ACKN CHECKSUM] [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a TCP/IP packet with source address SA, source port SP (if 0, a random one is generated for each packet), destination address DA, and destination port DP (if 0, a random one is generated for each packet), and control bits BITS. If SEQN and ACKN specified are non-zero, they are used. Otherwise, they are randomly generated for each packet. IP and TCP checksums are calculated if CHECKSUM is true; it is true by default. SEQN and ACKN should be in host order.

The RoundRobinTCPIPEncap element adds both a TCP header and an IP header.

You can give as many arguments as you'd like. Each argument specifies a single TCP/IP header. The element will cycle through the headers in round-robin order.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
RoundRobinTCPIPEncap(2.0.0.2 1.0.0.2 4 1022 1234 42387492 2394839 1,
                     2.0.0.2 1.0.0.2 2)
```

SEE ALSO

Strip(n), RoundRobinUDPIPEncap(n)

APPENDIX B

NAME

RoundRobinUDPIPEncap - Click element

SYNOPSIS

RoundRobinUDPIPEncap(SADDR DADDR [SPORT DPORT CHECKSUM?] [, ...])

PROCESSING TYPE

Agnostic

DESCRIPTION

Encapsulates each incoming packet in a UDP/IP packet with source address SADDR, source port SPORT, destination address DADDR, and destination port DPORT. The UDP and IP checksums are calculated if CHECKSUM? is true; it is true by default. If either DPORT or SPORT is 0, the port will be randomly generated for each packet.

The RoundRobinUDPIPEncap element adds both a UDP header and an IP header.

You can give as many arguments as you'd like. Each argument specifies a single UDP/IP header. The element will cycle through the headers in round-robin order.

The Strip(n) element can be used by the receiver to get rid of the encapsulation header.

EXAMPLES

```
RoundRobinUDPIPEncap(2.0.0.2 1.0.0.2 1234 1002 1,
                     2.0.0.2 1.0.0.2 1234)
```

SEE ALSO

Strip(n), UDPIPEncap(n)

APPENDIX B

SETSNIFFFLAGS(n) SETSNIFFFLAGS(n)

NAME

SetSniffFlags - Click element; sets sniff flags annotation.

SYNOPSIS

SetSniffFlags(FLAGS [, CLEAR])

PROCESSING TYPE

Agnostic

DESCRIPTION

Set the sniff flags annotation of incoming packets to FLAGS bitwise or with the old flags. if CLEAR is true (false by default), the old flags are ignored.

APPENDIX B

SETUDPTCPCHECKSUM(n) SETUDPTCPCHECKSUM(n)

NAME

SetUDPTCPChecksum - Click element

SYNOPSIS

SetUDPTCPChecksum()

PROCESSING TYPE

Agnostic

DESCRIPTION

Expects an IP packet as input. Calculates the ICMP, UDP or TCP header's checksum and sets the checksum header field. Does not modify packet if it is not an ICMP, UDP, or TCP packet.

SEE ALSO

SetIPChecksum(n)

APPENDIX B

STORESNIFFFLAGS(n) STORESNIFFFLAGS(n)

NAME

StoreSniffFlags - Click element; stores sniff flags annotation in packet

SYNOPSIS

StoreSniffFlags(OFFSET)

PROCESSING TYPE

Agnostic

DESCRIPTION

Copy the sniff flags annotation into the packet at offset OFFSET.

APPENDIX B

TCPMONITOR(n) TCPMONITOR(n)

NAME
TCPMonitor - Click element

SYNOPSIS
TCPMonitor()

PROCESSING TYPE
Push

DESCRIPTION
Monitors and splits TCP traffic. Output 0 are TCP traffic, output 1 are non-TCP traffic. Keeps rates of TCP, TCP BYTE, SYN, ACK, PUSH, RST, FIN, URG, and fragmented packets. Also keeps rates of ICMP, UDP, non-TCP BYTE, and non-TCP fragmented traffic.

ELEMENT HANDLERS
rates (read)
dumps rates

APPENDIX B

NAME

TCPSYNProxy - Click element

SYNOPSIS

TCPSYNProxy(MAX_CONNS, THRESHOLD, MIN_TIMEOUT, MAX_TIMEOUT [, PASSIVE])

PROCESSING TYPE

Push

DESCRIPTION

Help settup a three way TCP handshake from A to B by supplying the last ACK packet to the SYN ACK B sent prematurely, and send RST packets to B later if no ACK was received from A.

Expects IP encapsulated TCP packets, each with its ip header marked (MarkIPHeader(n) or CheckIPHeader(n)).

Aside from responding to SYN ACK packets from B, TCPSYNProxy also examines SYN packets from A. When a SYN packet from A is received, if there are more than MAX_CONNS number of outstanding 3 way connections per destination (daddr + dport), reject the SYN packet. If MAX_CONNS is 0, no maximum is set.

The duration from sending an ACK packet to B to sending a RST packet to B decreases exponentially as the number of outstanding connections to B increases pass 2^THRESHOLD. The minimum timeout is MIN_TIMEOUT. If the number of outstanding half-open connections is above 2^THRESHOLD, the timeout is

max(MIN_TIMEOUT, MAX_TIMEOUT >> (N >> THRESHOLD))

Where N is the number of outstanding half-open connections. For example, let the MIN_TIMEOUT value be 4 seconds, the MAX_TIMEOUT value be 90 seconds, and THRESHOLD be 3. Then when N < 8, timeout is 90. When N < 16, timeout is 45. When N < 24, timeout is 22 seconds. When N < 32, timeout is 11 seconds. When N < 64, timeout is 4 seconds. Timeout period does not decrement if the threshold is 0.

TCPSYNProxy has two inputs, three outputs. All inputs and outputs take in and spew out packets with IP header. Input 0 expects TCP packets from A to B. Input 1 expects TCP packets from B to A. Output 0 spews out packets from A to B. Output 1 spews out packets from B to A. Output 2 spews out the ACK and RST packets generated by the element.

If PASSIVE is true (it is not by default), monitor TCP three-way handshake instead of actively setting it up. In

this case, no ACK or RST packets will be sent. When an outstanding SYN times out, the SYN ACK packet is sent out of output port 2. No packets on port 0 are modified or dropped in this operating mode.

EXAMPLES

```
... -> CheckIPHeader() -> TCPSYNProxy(128,3,10,90) -> ...
```

ELEMENT HANDLERS

summary (read)

Returns number of ACK and RST packets sent and number of SYN packets rejected.

table (read)

Dumps the table of half-opened connections.

reset (write)

Resets on write.

SEE ALSO

MarkIPHeader(n), CheckIPHeader(n)

APPENDIX B

TCPSYNRESP(n) TCPSYNRESP(n)

NAME

TCPSYNResp - Click element

SYNOPSIS

TCPSYNResp()

PROCESSING TYPE

Push

DESCRIPTION

Takes in TCP packet, if it is a SYN packet, return a SYN ACK. This is solely for debugging and performance tunning purposes. No checksum is done. Spews out original packet on output 0 untouched. Spews out new packet on output 1.